CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Pseudotriángulos en Conjuntos de Puntos del Plano

Autor: Juan Carlos Caiza Román

Director: Gregorio Hernández Peñalver

MADRID, JULIO 2016

Agradecimientos

A Javier Prieto por ayudarme con la traducción en ciertos apartados.

A mis padres Esperanza y Juan y hermanos José y Noelia por todo el apoyo que he recibido de ellos en todo.

A mi tutor Gregorio Hernández por hacer esto posible y por facilitar en todo momento el correcto desarrollo de este trabajo.

ÍNDICE

1 RESUMEN

1.1 Resumen en Español

El objetivo del presente trabajo es la construcción de pseudotriángulos vacíos óptimos cuyos vértices pertenecen a un conjunto S de n puntos dados en el plano. Así, el objetivo será doble. Por una parte el estudio de dicha construcción y por otra el desarrollo de una aplicación de escritorio en java donde se van a implementar los algoritmos .

Un pseudotriángulo es un polígono simple con exactamente tres vértices convexos, los cuales están conectados por segmentos de líneas rectas o por cadenas de vértices cóncavos (consideramos que un vértice es cóncavo cuando su ángulo interior es mayor que π). El criterio de optimización va a depender de su área o perímetro.

Hay dos fases de construcción. Por un lado tenemos la construcción de un pseudotriángulo cuando los vértices convexos son dados y por otro cuando dichos vértices no son dados y, en este caso, tenemos que construir todos los pseudotriángulos que están contenidos en el conjunto P de n puntos.

Si el conjunto de puntos P está contenido dentro de los tres vértices convexos dados que forman el pseudotriángulo entonces el problema de construir un pseudotriángulo vacío puede ser resuelto en un tiempo $O(n^3)$ mientras que si no tenemos señalados los vértices, aplicamos el caso general y la construcción de todos los pseudotriángulos podría realizarse en un tiempo no superior a $O(n^6)$ para el problema de maximización y $O(n\,log\,n)$ para el de minimización.

Por otra parte, se ha desarrollado una aplicación en Java que integra los algoritmos estudiados. Además la interfaz incluye apartados de información donde se muestra en todo momento sobre lo que se está haciendo facilitando no sólo el uso de la misma si no también el aprendizaje y la comprensión del proceso de construcción y optimización. Para ello, se ha usado el entorno de programación desarrollado por JetBrains llamado IntelliJ debido a su gran integración con JavaFX Scene Builder para el diseño de la interfaz de usuario.

Finalmente, se han realizado una serie de pruebas para comprobar el correcto funcionamiento de la aplicación y, por tanto, de la correcta interpretación de los algoritmos descritos por primera vez en el artículo Optimal Empty Pseudo-Triangles in a Point Set de la Conferencia Canadiense de Geometría Computacional en agosto de 2009 y revisados posteriormente por Elsevier publicando dicha versión bajo el nombre “Empty Pseudo-Triangles in Point Sets” en 2011.

1.2 Resumen en Inglés

The purpose of this work is the construction of optimal empty pseudotriangles whose vertices belong to a set S of n points in the plane. Thus, there are two main objectives. On one hand, the analytical study of this construction and on the other hand the development of a desktop Java application in which the algorithms will be implemented.

A pseudotriangle is a simple polygon with three convex vertices connected either by straight line segments or by chains of concave vertices - we will consider a vertex to be concave when its interior angle is greater than π. The optimization criterion will depend on its area or perimeter.

We will deal with two cases. The first one is the construction of a pseudotriangle from a given set of points with its three convex vertices in it; the second one is the construction of every possible pseudotriangle from this set P of n points.

If the set of points P is contained within the three given vertices that make up the pseudotriangle, then the problem of building the empty pseudotriangle can be solved in $O(n^3)$ time, whereas if the vertices are not given, the general case has to be applied and the construction of all the pseudotriangles could be achieved in time no greater than $O(n^6)$ for the maximization and $O(n\ log\ n)$ for the minimization problems.

On the other hand, a Java application integrating the studied algorithms has been developed. The interface also includes information fields(???) where the current task is displayed at every time, making it easier not only to use but to learn and understand the process of construction and optimization. For this purpose a programming environment developed by JetBrains called IntelliJ has been used, due to its large integration with JavaFX Scene Builder for user interface design.

Finally, a series of tests have been performed in order to check the application works as expected and, thus, that the interpretation of the algorithms first described in the paper Optimal Empty Pseudo-Triangles in a Point Set from the Canadian Conference on Computational Geometry (Augost 2009) and checked afterwards by Elsevier publishing "Empty Pseudo-Triangles in Point Sets" in 2011.

2 INTRODUCCIÓN Y OBJETIVOS

2.1 Introducción

Para hablar sobre la introducción del proyecto, es necesario volver a enunciar el concepto de pseudotriángulo ya mencionado en el resumen y ampliarlo un poco más. Un pseudotriángulo es un polígono simple con exactamente tres vértices convexos, los cuales están conectados por segmentos de líneas rectas o por cadenas de vértices cóncavos (consideramos que un vértice es cóncavo cuando su ángulo interior es mayor que π). Por definición, cualquier triángulo es un pseudotriángulo y el cierre convexo de cualquier pseudotriángulo es un triángulo. Un pseudotriángulo es estrellado si existe un punto interior q tal que para cualquier punto p en el pseudotriángulo, el segmento $\overline{pq}$ está contenido dentro también.

Los pseudotriángulos fueron introducidos en el contexto de relaciones de visibilidad computacional entre obstáculos convexos en el plano en 1996 pero fue más tarde cuando un número de problemas de optimización diferentes de pseudotriángulos, como por ejemplo la descomposición de una región en pseudotriángulos, surgieron y fueron estudiados. Este trabajo se centra en pseudotriángulos vacíos, es decir, si no contienen puntos en su interior, definidos por un conjunto P dado de n puntos en $\mathbb{R}^2$.

En particular, el problema a tratar es la minimización del perímetro o la maximización del área de un pseudotriángulo vacío. Al principio trataremos este problema cuando los tres vértices convexos sean dados y después, se considerarán dichas optimizaciones para todos los posibles pseudotriángulos contenidos en un conjunto de puntos.

Las triangulaciones de puntos en el plano se generalizan de forma natural a pseudotriangulaciones. Una pseudotriangulación de un conjunto S de puntos del plano es una partición del cierre convexo de S en pseudotriángulos cuyo conjunto de vértices es exactamente S. Estas estructuras han adquirido relevancia en los últimos 20 años por sus aplicaciones a, por ejemplo, la planificación de movimientos y la teoría de la rigidez.

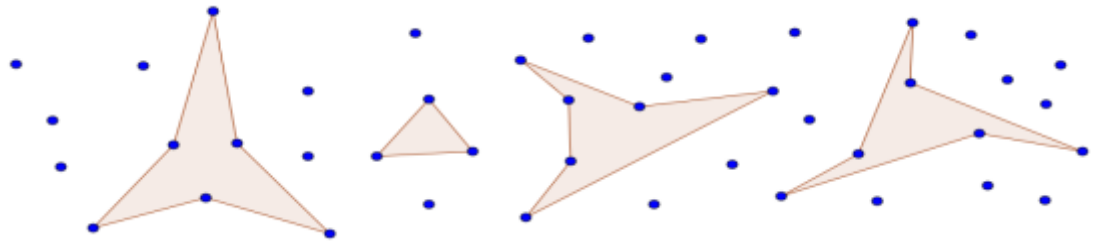


Imagen 1 - Pseudotriángulos en un conjunto de puntos

2.2 Objetivos

A continuación se listan todos los objetivos que se han programado para el correcto desarrollo de este trabajo.

- Estudiar las diferentes pseudotriangulaciones de un conjunto de puntos en el plano para posteriormente estudiar los algoritmos que permiten construir un pseudotriángulo con vértices en un conjunto de puntos optimizando algún parámetro.
- Estudio de las diferentes pseudotriangulaciones de un conjunto de puntos en el plano.
- Estudio de los algoritmos que permiten construir un pseudotriángulo con vértices en un
- Conjunto de puntos optimizando algún parámetro.
- Diseño de la aplicación (estructura general, estructuras de datos, algoritmos)
- Programación de la aplicación.
- Verificación y prueba con datos reales.
- Redacción de la memoria y presentación de resultados.

3 ESTADO DEL ARTE: PROBLEMA GEOMÉTRICO

En este apartado vamos a estudiar el problema que se nos plantea al construir pseudotriángulos maximizando o minimizando su área o perímetro.

3.1 Preliminares

Antes de centrarnos en el problema geométrico, procedemos a definir ciertos conceptos y asumir ciertas condiciones que nos ayudarán posteriormente a enunciar el problema con propiedad y a entender cómo se ha tratado:

- Sean u, v y w tres puntos en el plano, asumimos que el segmento que conecta u y v es horizontal con u a la izquierda de v y que w se sitúa por encima de dicho segmento.
- Sea P un conjunto de n puntos contenidos dentro del triángulo Δuvw, se asume que no hay tres puntos contenidos en $P \cup \{u, v, w\}$ que pertenezcan a la misma línea.

- Sean dos puntos p y q, denotamos la línea que pasa por ambos como $l(p,q)$.
- Cualquier pseudotriángulo con u, v y w como vértices convexos tiene cadenas cóncavas que conectan dichos vértices dos a dos por lo que cada vértice convexo es el punto de encuentro entre dos cadenas cóncavas.
- Sean pi, pj y pk los primeros vértices encontrados al recorrer las cadenas cóncavas en sentido de las agujas del reloj a lo largo del pseudotriángulo desde u, v y w respectivamente.

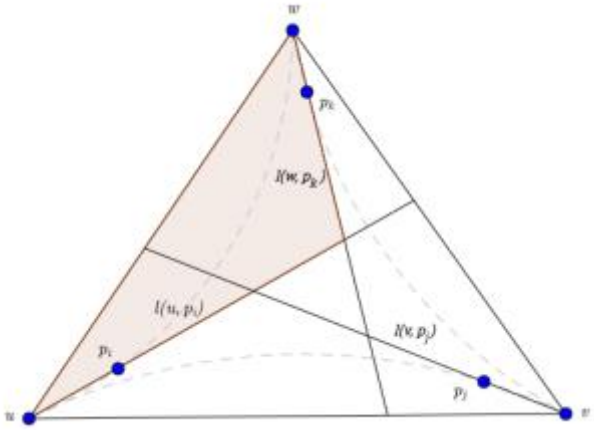


Imagen 2 - Cada Pseudotriángulo vacío es definido por una única terna de puntos pi, pj y pk.

La imagen 2 muestra estos vértices y tres líneas, las cuales contienen una esquina y su siguiente vértice en sentido de las agujas del reloj. Además, hay una zona sombreada que está limitada por las líneas $l(u,w)$, $l(u,p_i)$ y $l(w,p_k)$. Para que el pseudotriángulo sea vacío, todos los puntos de P dentro de dicha zona deben estar rodeados por la cadena cóncava que conecta u con w. Por lo tanto, la cadena poligonal que rodea dichos puntos se define únicamente por p_i y p_k y la denotamos como $C_{uw}(i,k)$. Análogamente, la cadena cóncava $C_{vw}(k,j)$ conecta w con v está únicamente definida por p_k y p_j y la cadena cóncava $C_{uv}(j,i)$ que conecta v con u está únicamente definida por p_j y p_i. En consecuencia, observamos lo siguiente.

Observación 1. Tres puntos pi, pj y pk determinan un pseudotriángulo vacío sí y sólo sí:

(i) El punto pi está situado a la izquierda de $l(w,pk)$, pj está debajo de $l(u,pi)$ y pk arriba de $l(v,pj)$.

(ii) La región triangular situada debajo de $l(u,pi)$, encima de $l(v,pj)$ y a la izquierda de $l(w,pk)$ no contiene ningún punto en su interior.

Además, dicho pseudotriángulo será único.

A partir de aquí simplificaremos la notación de las cadenas cóncavas omitiendo los índices de las cadenas si los puntos pi, pj y pk están definidos en el contexto. Por ello, las cadenas serán denotadas como C_{uw}, C_{vw} y C_{uv}.

3.2 Pseudotriángulos vacíos

En esta sección se estudia el problema algorítmico de construir un pseudotriángulo que sea óptimo con respecto a su perímetro o su área. Se considera la minimización y maximización para cada uno. En primer lugar se estudiará el caso para cuando los tres vértices son dados y posteriormente para cuando no son dados.

3.2.1 Pseudotriángulos vacíos optimizados con los vértices convexos dados

Dados tres puntos u, v y w y un conjunto P de n puntos dentro de Δuvw, mostramos cómo determinar un pseudotriángulo optimizado que tiene u, v y w como vértices convexos, puntos de P como vértices cóncavos y ningún punto de P en el interior del triángulo.

En Observación 1 afirmamos que tres puntos pi, pj y $pk \in P$ definen como mucho a un pseudotriángulo si asumimos que los segmentos $\overline{up_i}, \overline{vp_j}$ y $\overline{wp_k}$ son las primeras aristas de las cadenas C_{uv}, C_{vw} y C_{uw}, respectivamente. Esta observación nos lleva inmediatamente a un tiempo algorítmico de $O(n^4)$: ordenamos los puntos según su coordenada x, enumeramos todas las ternas de puntos p_i, p_j y p_k y determinamos si existe un pseudotriángulo vacío para dicha terna. Cada pseudotriángulo puede ser generado en tiempo lineal gracias al método de Graham debido a la previa ordenación de puntos. También, debido a dicha ordenación, podemos determinar si un pseudotriángulo es vacío en tiempo lineal si simultáneamente hacemos un barrido sobre el conjunto de puntos P y sobre el pseudotriángulo. Finalmente, el área o el perímetro pueden ser calculados en tiempo lineal también.

Podemos mejorar este método para que se ejecute en tiempo $O(n^3)$ enumerando todas las posibilidades de una forma más eficiente. Básicamente, fijaremos p_i y p_j y trataremos todas las posibilidades para p_k todo a la vez en tiempo lineal. Esto es posible porque el cambio de la suma total de los pseudotriángulos consecutivos que testeamos es lineal en lugar de cuadrático. En nuestra descripción ignoramos el caso dónde una de las cadenas es

sólo una línea de segmentos que conectan dos de los vértices de u, v y w porque este caso especial es más fácil que el caso general y podemos tratarlo a parte.

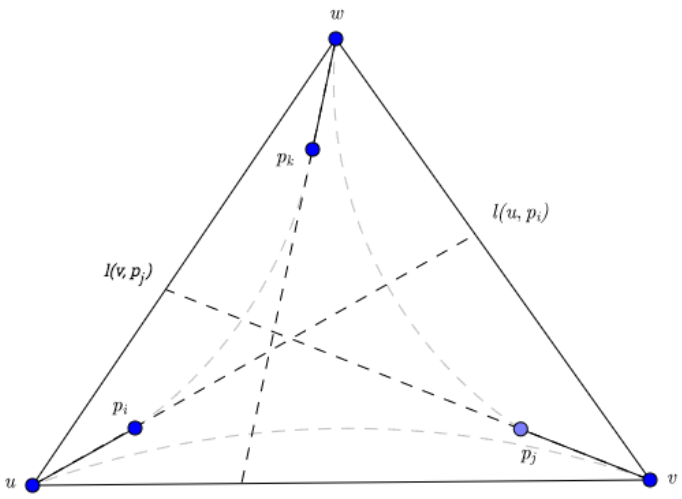


Imagen 3 - Los puntos pi, pj y pk y las cadenas implicadas.

Para continuar con la optimización, vamos a cambiar alguna condición y asumir unas cuántas condiciones:

- Fijamos p_i y p_j y asumimos que el segmento $\overline{up_i}$ pertenece a la cadena C_{uw} y que el segmento $\overline{vp_j}$ pertenece a C_{vw} (véase la imagen 3).
- Todos los puntos de P estrictamente debajo de $l(u, p_i)$ y $l(v, p_j)$ deben estar en o por debajo de la cadena C_{uv}, por tanto C_{uv} queda completamente definida.
- Sea P' un subconjunto de puntos de P situados por encima de $l(u, p_i)$ o $l(v, p_j)$. Dicho conjunto ordenado en sentido contrario a las agujas del reloj alrededor de w será denotado como $p_1, \dots, p_m$.
- Para cualquier pseudotriángulo vacío con p_i y p_j siguiendo las especificaciones anteriores, existe un punto p_k tal que todos los puntos a la izquierda de o sobre $l(w, p_k)$ están a la izquierda de C_{uw} y todos los puntos a la derecha de o sobre $l(w, p_{k+1})$ están a la derecha de C_{vw}. Este pseudotriángulo vacío incluye los lados $\overline{wp_k}$ en C_{uw} y $\overline{wp_{k+1}}$ en C_{vw}. Entonces, observamos que $i \leq k < k+1 \leq j$ ya que de otra forma el pseudotriángulo no sería vacío o autointersecado o alguna de sus cadenas tendría un vértice convexo. Además, si p_i se sitúa sobre $l(v, p_j)$ o p_j se sitúa

sobre $l(u, p_i)$, entonces cierta región triangular en Δuvw no debe contener ningún punto de P ya que, de otra forma, según nuestras especificaciones, no podríamos construir un pseudotriángulo vacío.

Concluimos entonces que hay linealmente muchas opciones para la cadena C_{uw} que incluye $\overline{up_i}$ y C_{uw} queda determinado si fijamos p_k. En otras palabras, C_{uw} es el límite del cierre convexo de u, w, p_i, p_k y los puntos de P' que están a la izquierda de $l(w, p_k)$ cuando excluimos el lado $\overline{uw}$.

A continuación presentamos un algoritmo donde en un conjunto de puntos P comprobamos si p_i y p_j son puntos válidos para construir un pseudotriángulo vacío y optimizado en términos de área o perímetro. Se usa $l(w,p)^{left}$ y $l(w,p)^{right}$ para denotar el semiplano que está a la izquierda o la derecha de $l(w,p)$, respectivamente. También se usa $l(u,p)^-$, $l(u,p)^+$, $l(v,p)^-$ y $l(v,p)^+$ para los semiplanos superiores o inferiores a las correspondientes líneas.

Algorithm *Test-Pair*
Input: A set P of n points inside Δuvw, and two points $p_i, p_j \in P$.
Output: TRUE if and only if some empty pseudo-triangle exists with $\overline{up_i}$ as an edge of C_{uw} and $\overline{vp_j}$ as an edge of C_{vw}.

1. **if** $\ell(w, p_i) \cap \overline{vp_j} \neq \emptyset$ or $\ell(w, p_j) \cap \overline{up_i} \neq \emptyset$ **return** FALSE
2. **if** $\ell(u, p_i)^- \cap \ell(v, p_j)^+ \cap \ell(w, p_i)^{\text{left}} \cap P \neq \emptyset$ **return** FALSE
3. **if** $\ell(u, p_i)^+ \cap \ell(v, p_j)^- \cap \ell(w, p_j)^{\text{right}} \cap P \neq \emptyset$ **return** FALSE
4. **return** TRUE

Imagen 4 - Algoritmo Test-Pair para encontrar los puntos pi y pj.

Además de la condición $i \leq k < k+1 \leq j$, también es necesario que todos los puntos $p_i, \ldots, p_k$ estén en o sobre $l(w, p_i)$, de otra forma la cadena cóncava C_{uw} existe con p_i y p_k en ella. Simétricamente, $p_{k+1}, \ldots, p_j$ deben situarse en o sobre $l(v, p_j)$ para tener una cadena cóncava con C_{vw} con p_j y p_{k+1} en ella. Consideramos todas las opciones de $p_k \in P'$ que puedan dar como resultado una cadena C_{uw} válida con p_i y p_k en ella. La unión de estas cadenas de C_{uw} sobre las posibles k válidas es un grafo G. La adición de una línea de G es planar porque para todas las $k' < k$, el conjunto de puntos $(l(u,p_i)^+ \cap l(w,p_{k'})^{left} \cap P') \cup \{u, w, p_i, p_{k'}\}$ está contenido en el conjunto de puntos $(l(u,p_i)^+ \cap l(w,p_k)^{left} \cap P') \cup \{u, w, p_i, p_k\}$ y por lo tanto los límites del cierre convexo no puede tener lados que se intersequen.

Si borramos w y todos los lados que inciden en él, obtenemos un árbol $T_i(u)$ siendo u su raíz. Dicho árbol puede ser procesado con un simple algoritmo incremental.

Algorithm *Rooted-Tree*
Input: Vertex u, point set P' sorted counterclockwise around w, and points $p_i, p_j \in P'$.
Output: A tree $T_i(u)$.

```
1.  T_i(u) ← u p_i
2.  p ← u and q ← p_i
3.  k ← i + 1
4.  while k < j and p_k is above ℓ(u, p_i)
5.        γ ← the ray directed from p towards q
6.        while p_k is to the right of γ
7.              q ← p and p ← the parent of p in T_i(u)
8.              γ ← the ray directed from p towards q
9.        T_i(u) ← T_i(u) ∪ p_k q
10.       p ← q and q ← p_k and k ← k + 1
11.  return T_i(u)
```

Imagen 5 - Algoritmo Rooted-Tree para encontrar el árbol cuya raíz puede ser u o v.

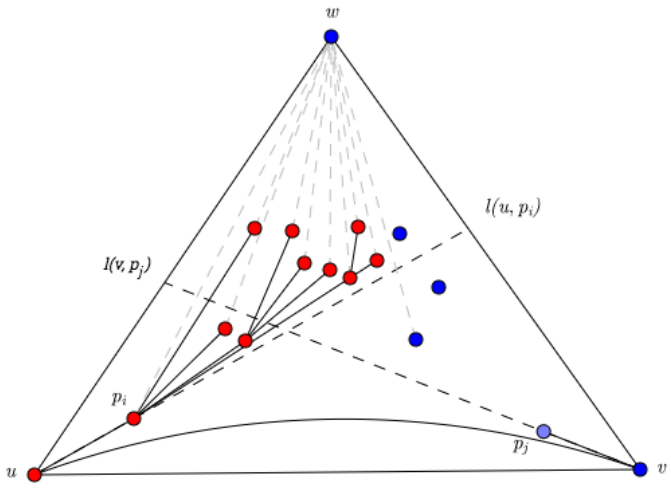


Imagen 6 - Árbol generado por Rooted-Tree con raíz en u.

Lema 1. El algoritmo Rooted-Tree computa correctamente $T_i(u)$.

Demostración. El algoritmo itera sobre los p_k puntos empezando desde p_{i+1} hasta p_j o hasta que se encuentre un punto por debajo de $l(u, p_i)$. Sea $T_i^k(u)$ el árbol con todos los lados elegidos después de tratar p_k, probamos por inducción en $k \geq i$ que $T_i^k(u)$ es la unión de todas las cadenas C_{uw} definidas por p_i y una de $p_i, \dots, p_k$ menos los lados $\overline{wp_i}, \dots, \overline{wp_k}$, lo cual significa que $T_i^k(u)$ es correcto. Cuando $k = i$, empezamos ciertamente con $T_i^i(u) = \overline{up_i}$ por lo que esto es correcto. Ahora suponemos que el árbol $T_i^{k-1}(u)$ es correcto para $k > i$ y consideramos el siguiente punto en sentido contrario a las agujas del reloj es p_k. Ya que el resultado de incluir p_k en el árbol sería $T_i^k(u)$, deberíamos añadir exactamente un lado y este lado es parte del límite del cierre convexo de $(l(u, p_i)^+ \cap l(w, p_k)^{left} \cap P') \cup \{u, w, p_i, p_k\}$. Ya que p_k es el más cercano a w en sentido contrario a las agujas del reloj, queda claro que p_k forma parte del cierre convexo $(l(u, p_i)^+ \cap l(w, p_k)^{left} \cap P') \cup \{u, w, p_i, p_k\}$. Uno de los lados del cierre convexo es $\overline{wp_k}$ gracias a la ordenación en sentido contrario a las agujas del reloj alrededor de w. El otro lado del cierre convexo es parte de la otra tangente a través de p_k del cierre convexo del anterior paso. La forma en la que este lado es encontrado por el algoritmo es exactamente igual al paso del método de Graham para computar el cierre convexo de un conjunto de punto. Por tanto, el lado correcto desde p_k es añadido y el árbol $T_i^k(u)$ es, así mismo, correcto.

Asumimos que estamos interesados en el perímetro mínimo de un pseudotriángulo vacío. Desde la raíz hace las hojas, calculamos y almacenamos con cada nodo (punto) la distancia Euclídea del camino desde ese nodo a u. Para cada nodo, entonces, añadimos la distancia Euclídea desde ese nodo hasta w. Haciendo esto, tenemos la longitud de la cadena cóncava C_{uw} almacenada con cada nodo, si ese nodo es elegido como p_k.

Un algoritmo similar construye el árbol $T_j(v)$ que consiste en los lados de las cadenas de C_{vw} que terminan con $\overline{vp_j}$. Otra vez, computamos y almacenamos las longitudes de las cadenas con cada nodo. Si ese nodo es elegido como p_{k+1}.

No todos los puntos de P' son usados en ambos árboles. Sea j' tal que $p_{j'}$ es el punto de P' con el índice más bajo que está por debajo de $l(u, p_i)$, si tal punto no existe, entonces $j' = j$. Así entonces $p_{j'}$ es el punto donde la construcción de $T_i(u)$ se para. Igualmente, sea i' tal que $p_{i'}$ es el punto de P' con el índice más alto que está por debajo de $l(v, p_j)$ o $i' = i$. Observamos el siguiente lema.

Lema 2. Un pseudotriángulo vacío existe con $\overline{wp_k}$ en C_{uw} y $\overline{wp_{k+1}}$ en C_{vw} si y sólo si $i' \leq k < k+1 \leq j'$.

Por tanto, nuestro algoritmo es como se describe a continuación.

Algorithm *Shortest-Pseudo-Triangle*
Input: A set P of n points inside a triangle Δuvw.
Output: An empty pseudo-triangle with minimum perimeter

```
1.  Sort P counterclockwise in angular order around w
2.  for all choices of p_i and p_j for which Test-Pair returns TRUE
3.      Select the subset of points below ℓ(u, p_i) and below ℓ(v, p_j) and compute C_uv
4.      Determine P' and compute T_i(u) and T_j(v) with Rooted-Tree
5.      for k ← i' to j' − 1
6.          Locate p_k in T_i(u) and p_{k+1} in T_j(v) and add the stored values,
7.          maintaining the minimum sum found so far
8.  return the minimum perimeter empty pseudo-triangle
```

Imagen 7 - Algoritmo Shortes Pseudotriangle para encontrar el pseudotriángulo vacío optimizado respecto de algún parámetro.

Analizamos el tiempo de ejecución del algoritmo Shortest-Pseudo-Triangle. El bucle externo recorre $O(n^2)$ posibilidades. Test-Pair se ejecuta en tiempo lineal. C_{uv} puede ser computado en tiempo lineal usando el conjunto de puntos ordenado alrededor de w ignorando aquellos puntos que están por encima de $l(u, p_i)$ o por encima de $l(v, p_j)$. La secuencia ordenada para P' puede ser extraída de la secuencia ordenada de P en tiempo lineal también. La localización de p_k en $T_i(u)$ y p_{k+1} en $T_j(v)$ puede ser encontrada en tiempo lineal en conjunto por el recorrido de sus previas ubicaciones (cada arista del árbol es atravesada como mucho dos veces).

Falta por mostrar que el árbol $T_i(u)$ y $T_j(v)$ puede ser computado en tiempo lineal también. El único momento cuando la adición del siguiente punto p_k pueda tardar en ejecutarse más que un tiempo lineal es cuando p_k está a la derecha de γ varias veces consecutivas. Sin embargo, cada vez, hay un nodo del árbol $T_i(u)$ que no será encontrado otra vez en el resto del algoritmo (denotado como q). Por consiguiente, el coste general de estos pasos es lineal. Podemos calcular todos los tamaños de las aristas en tiempo lineal usando un simple árbol transversal desde la raíz u hasta las hojas.

Las adaptaciones necesarias para calcular el máximo perímetro, área mínima o área máxima de un pseudotriángulo vacío son sencillas y fáciles de entender. Si estamos interesados en calcular el área, podemos almacenar para cada nodo el área del cierre convexo del camino desde ese nodo hasta la raíz en lugar de la longitud de ese camino.

Teorema 1. Dados n puntos contenidos en un triángulo, un pseudotriángulo con el mínimo o máximo perímetro o con la mínima o máxima área pueden ser calculados en tiempo $O(n^3)$ usando un espacio lineal.

3.2.2 Pseudotriángulos optimizados en un conjunto de puntos

Ahora discutimos el caso donde los tres vértices convexos no son proporcionados de antemano pero pueden ser elegidos libremente dentro del conjunto P de n puntos en el plano.

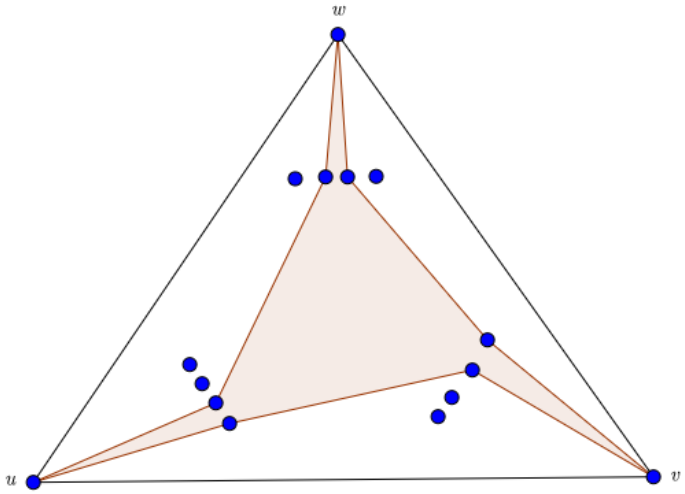


Imagen 8 - Cota mínima en el máximo número de pseudotriángulos vacíos.

Teorema 2. Dados tres puntos u, v y w, cualquier set de n puntos dentro de Δuvw define $\Omega(n^2)$ pseudotriángulos vacíos con u, v y w como vértices convexos y esta cota es la mejor posible.

Demostración. Hay $6 * \binom{n}{3}$ posibilidades para p_i, p_j y p_k por tanto, la cota superior sigue lo dicho en la **Observación 1** y, como podemos observar en la imagen 8, no hay otra mejor cota superior posible usando dicha construcción.

Teorema 3. Cualquier conjunto de n puntos en el plano define $O(n^6)$ pseudotriángulos vacíos y esta cota es la mejor posible.

Demostración. Sea P un conjunto de n puntos en el plano, la cota superior surge de tomar todas las combinaciones de puntos tomados de tres en tres de P como u, v y w y usando el resultado del **Teorema 2**. La rigidez que surge de tomar la construcción usada en el **Teorema 1** usando sólo $n/2$ puntos dentro de Δuvw y reemplanzando u, v y w por $n/6$ puntos para cada uno. w es reemplazada por $n/6$ puntos en una línea horizontal estrechamente espaciada y con respecto a w. Similarmente, u y v son reemplazados por $n/6$ puntos en cada uno y sobre líneas que forman ángulos de $60°$ (para v) y $-60°$ (para u) con el eje x.

Obtenemos que el máximo número posible de pseudotriángulos vacíos en este caso es $\Theta(n^6)$. Esto nos lleva a una adaptación sencilla de nuestro algoritmo para este caso: tomamos simplemente todas las ternas de tres puntos posibles de P como vértices convexos de un pseudotriángulo y aplicamos el algoritmo. Esto obviamente incrementa el tiempo de ejecución por un factor de $O(n^3)$ sin influenciar el requisito de espacio.

Teorema 4. Dados n puntos en el plano, un pseudotriángulo con el mínimo o máxima área o mínimo o máximo perímetro puede ser calculado en tiempo $O(n^6)$ usando un espacio lineal.

Sin embargo, con una simple observación podemos drásticamente reducir los tiempos de ejecución para los dos problemas de minimización como mostraremos a continuación. Empezamos con la minimización del perímetro.

Lema 2. Dado un conjunto de puntos en el plano, cualquier pseudotriángulo vacío con el perímetro mínimo o área mínima es un triángulo.

Prueba. Sea ך el perímetro mínimo de un pseudotriángulo con más de tres vértices en sus límites. Podemos triangular ך y obtener triángulos vacíos en ך con un perímetro más pequeño, lo cual nos lleva a una contradicción. El mismo argumento se aplica para el área mínima de un pseudotriángulo.

Dobkin al demuestra que el perímetro mínimo de un triángulo puede ser determinado en un tiempo $O(n \log n)$ y por tanto un pseudotriángulo vacío con un perímetro mínimo puede ser calculado en tiempo $O(n \log n)$. Para minimizar el área podemos usar el resultado de Edelsbrunner y Guibas y podemos concluir que un pseudotriángulo vacío con el área mínima puede ser calculado en un tiempo $O(n^2)$.

3.3 Aplicaciones de Pseudotriangulaciones.

A continuación se van a enunciar varios ejemplos en los que dicho estudio es importante. Cabe destacar que los ejemplos han sido extraídos del *survey* Pseudo-Triangulations, de Günter Rote, Francisco Santos e Ileana Streinu.

3.3.1 Triangulaciones geodésicas equilibradas para disparos de rayos.

Las triangulaciones geodésicas equilibradas fueron introducidas como estructuras de datos que se encargaban del Ray casting y de búsqueda de camino más corto en un grafo planar que va cambiando dinámicamente.

Ray casting se refiere al problema de encontrar el primer punto límite que es golpeado por un rayo de consulta. El enfoque clásico es que un polígono P de n vértices usa una triangulación de P. Después de localizar el punto de inicio del rayo de consulta en uno de los triángulos, el siguiente sigue el rayo desde el triángulo hacia el triángulo adyacente hasta que el límite es golpeado. El tiempo de ejecución, después del paso de encontrar el punto inicial, es proporcional al número de triángulos que se atraviesan, los cuales pueden ser a lo sumo $O(n)$, dependiendo de la triangulación. Así mismo, el camino más corto entre dos puntos consultados puede ser encontrado fácilmente después de identificar la secuencia única de los triángulos que conectan los dos triángulos que contienen los puntos consultados.

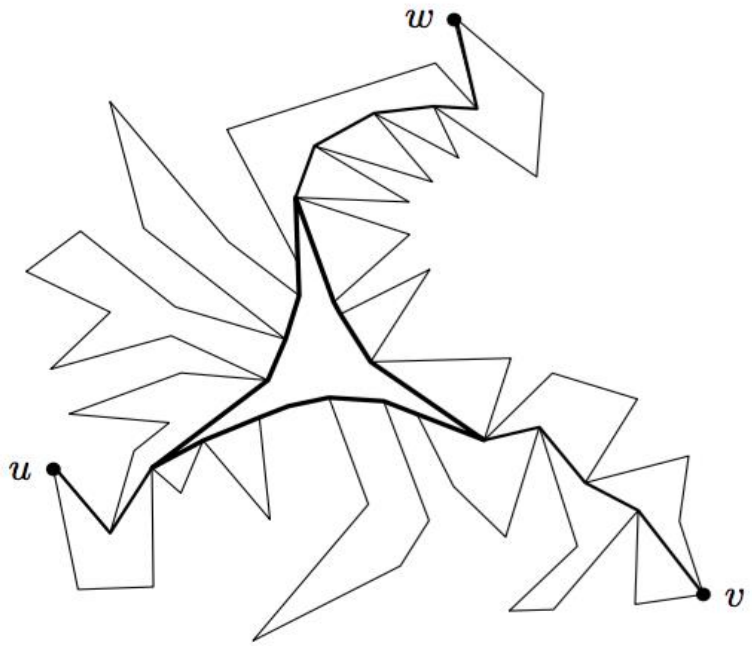


Imagen 9 - Triangulación geodésica

El objetivo es que el camino esté formado por el menor número de triángulos posible y que, a la vez, se mantenga la estructura buscada bajo los cambios que pueda experimentar el polígono. Para ello, se usa triangulaciones geodésicas en lugar de triangulaciones. Tres caminos geodésicos entre uv, vw y uw forman un triángulo geodésico donde tiene un pseudotriángulo en el centro (también conocido como región deltoidal) y posiblemente algunos caminos complejos, lo cuales son compartidos entre dos caminos geodésicos. Véase la imagen 9. Sea $\hat{P}$ un polígono de n vértices convexos los cuales corresponden a los vértices de P así como aparecen en la frontera. Consideramos una triangulación $\hat{T}$ de $\hat{P}$. Para cada triángulo $\hat{u}\hat{v}\hat{w}$ en $\hat{T}$, podemos considerar el correspondiente triángulo geodésico uvw en $\hat{P}$. El conjunto de estos triángulos geodésicos formarán una triangulación geodésica de P. Véase la imagen 9. Es una pseudotriangulación del interior de P pero almacena información adicional sobre la correspondencia entre aristas y sus caminos iniciales y sobre las adyacencias en la triangulación $\hat{T}$. La triangulación $\hat{T}$ del polígono convexo de n vértices P puede ser representado como un árbol binario (después de seleccionar una arista como "raíz"). Para seguir con el paradigma mencionado arriba para el *disparo de rayos*, uno tiene que atravesar la secuencia geodésica de triángulos. El ir de un triángulo geodésico a otro adyacente no volverá a ser una operación que se ejecute en tiempo constante porque

un triángulo geodésico no tiene un tamaño constante pero puede ser llevado a cabo en tiempo logarítmico con las estructuras de datos apropiadas.

La ventaja de usar triángulos geodésicos es que será más fácil obtener una cota en el número de triángulos geodésicos atravesados. Si la triangulación $\hat{T}$ está balanceada en el sentido de que está construida dividiendo siempre la parte restante de la frontera en partes iguales, el número de triángulos en cualquier camino entre dos triángulos es $O(\log n)$ (el cual es claramente el mejor posible).

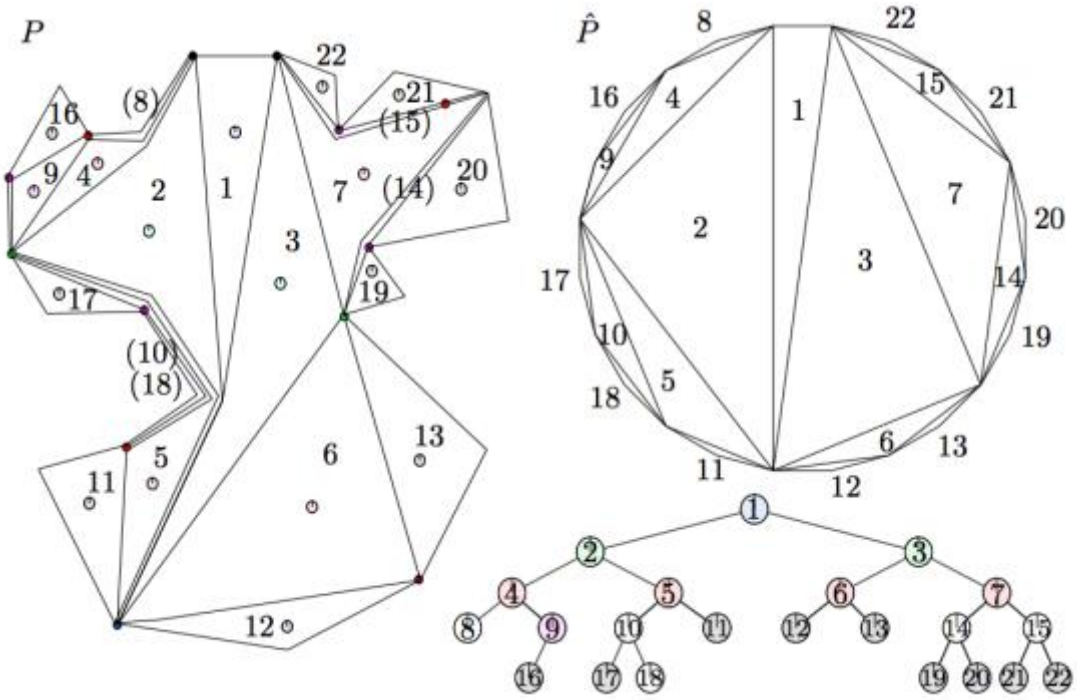


Imagen 10 - Resultado final

3.3.2 Pseudotriangulaciones de obstáculos convexos.

En contraste con la sección anterior, la cual lidia con conjuntos de puntos, aquí consideramos una colección de n obstáculos convexos disjuntos en el plano y se definen sus pseudotriangulaciones más abajo. Para simplificar, asumimos que los cuerpos son regulares. En este escenario un pseudotriángulo no es un polígono. Es una región limitada por una curva de Jordan que consiste en el tres piezas cóncavas hacia dentro dónde cada dos se juntan en una cúspide con una tangente común. La imagen muestra una pseudotriangulación de tres cuerpos convexos dónde aparecen cuatro pseudotriángulos.

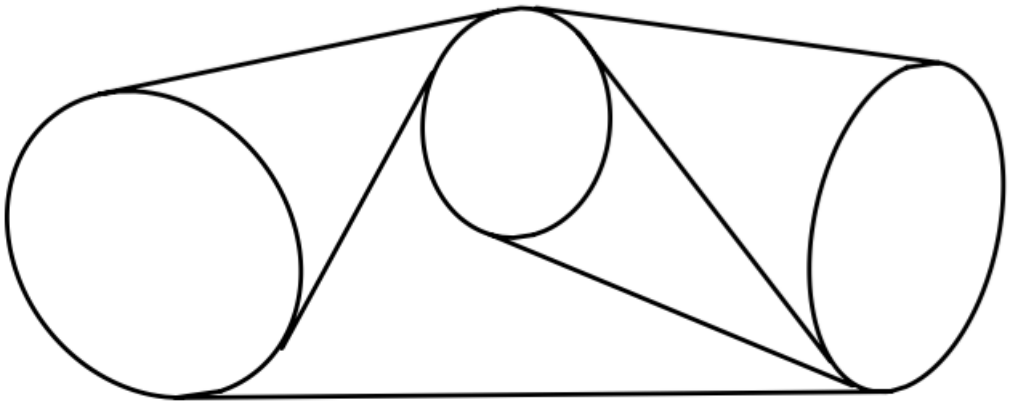


Imagen 11 - Pseudotriangulaciones de obstáculos convexos

El término análogo de la asunción de la posición general es que no hay tres obstáculos que tengan una línea tangente común. El segmento entre dos puntos de tangencia se llama *bitangencia*. Decimos que una bitangencia es *libre* si no se interseca con algún otro obstáculo.

Una pseudotriangulación para un conjunto convexo de obstáculos es un conjunto maximal de bitangentes que no se cruzan (imagen 11). Estas pseudotriangulaciones tienen propiedades análogas a las pseudotriangulaciones de conjuntos de puntos como que cada punto bitangente interior en una pseudotriangulación puede ser volteado, etc.

A parte de las aplicaciones de visibilidad de la siguiente sección, Pocchiola y Vegter mostraron después que el problema de encontrar un recubrimiento poligonal de una colección de cuerpos disjuntos convexos es equivalente a encontrar una pseudotriangulación de ellos.

3.3.3 El problema de la visibilidad.

El problema de la visibilidad fue introducido por Pcchiola y Vegter. Consideramos un conjunto dirigido de *segmentos visibles* o segmentos *maximales libres*, los cuales no intersecan con algún obstáculo en su interior pero que no pueden ser extendidos sin cortar un obstáculo. Ver imagen 12. Tal segmento empieza y termina en un obstáculo o se extiende hasta el infinito en una o dos direcciones. Los segmentos pueden ser movidos continuamente, formando un espacio topológico, el problema de la visibilidad. Este espacio

es bidimensional ya que un segmento puede estar parametrizado localmente por la pendiente y la distancia signada desde el origen. Todos los segmentos pueden ser transformados en otros del mismo conjunto mientras se mantienen sus puntos finales en los mismos dos obstáculos desde una cara bidimensional del problema de la visibilidad. Un segmento alcanza el límite de una cara cuando éste se convierte en una tangente a algún objeto. Una arista de la visibilidad compleja está, por tanto, formada por un segmento libre tangente a un objeto y que rota alrededor del mismo mientras se mantiene su punto inicial y su terminal en otros dos objetos. Finalmente, un vértice de visibilidad compleja corresponde a un bitangente libre, en el contexto de la sección previa.

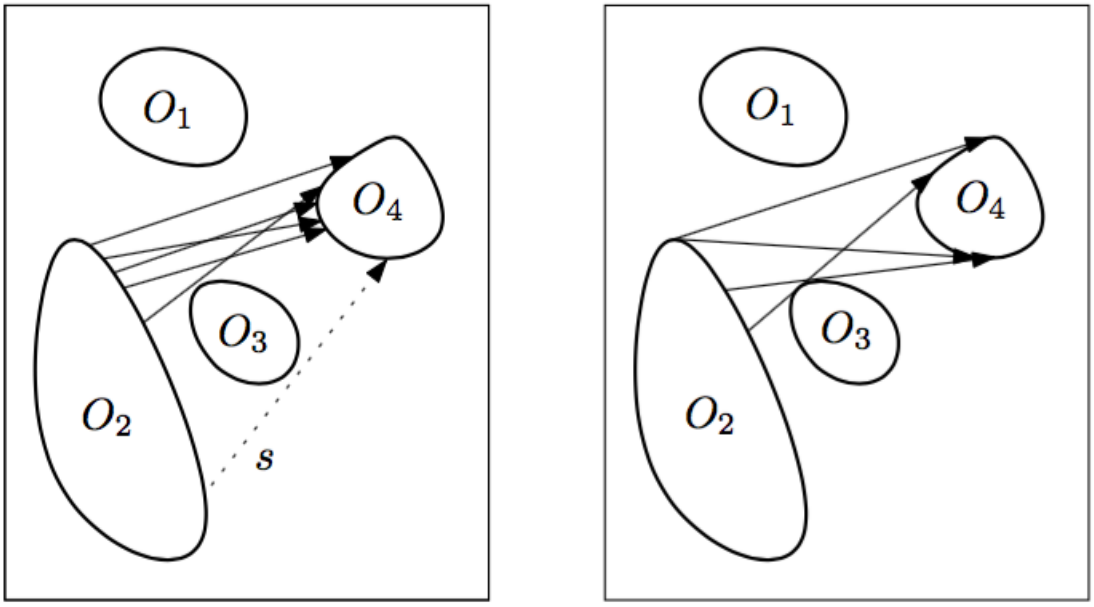


Imagen 12 - Problema de la visibilidad

4 DESARROLLO

4.1 Descripción

En esta sección se describirá todo lo relativo a la programación de la aplicación. Desde las herramientas utilizadas hasta el manual de usuario. También se hablará de los problemas encontrados, entre otras cosas.

4.2 Herramientas utilizadas

En este apartado nos centraremos en la enumeración y descripción de las características de las tecnologías software utilizadas en la realización del trabajo.

4.2.1 JavaFX

JavaFX es un conjunto de paquetes multimedia y de gráficos que permite a los desarrolladores diseñar, crear, testear, depurar y desplegar aplicaciones cliente muy sofisticadas para operar consistentemente a través de diversas plataformas.

Debido a que la librería de JavaFX está escrita como una API de Java, el código de una aplicación JavaFX puede referenciar APIs desde cualquier librería de Java. Por ejemplo, una aplicación JavaFX puede usar las librerías API de Java para acceder a los recursos del sistema nativamente y conectarse a las aplicaciones middleware basadas en servidor.

La apariencia y la interacción de las aplicaciones de JavaFX pueden ser personalizadas. Las hojas de estilo en cascada (CSS) separan la apariencia y estilo de la implementación para que los desarrolladores puedan centrarse en la programación. Los diseñadores gráficos pueden fácilmente personalizar la apariencia y el estilo de la aplicación a través de CSS. El desarrollo de la interfaz de usuario (UI) también puede realizarse sin la necesidad de programar ya que Scene Builder se encarga del etiquetado FXML que puede ser portado a un entorno de desarrollo integrado (IDE) para que los desarrolladores puedan añadir la lógica de la aplicación en Java.

En 2007 Sun Microsystems anunció por primera vez dicha plataforma en la conferencia mundial de desarrolladores en Java llamada JavaOne. JavaFX es una plataforma software para la creación de aplicaciones de escritorio así como aplicaciones enriquecidas en internet (Rich Internet Applications, (RIAs) las cuales pueden ejecutarse en una gran variedad de dispositivos.

El propósito de JavaFX fue reemplazar a Swing como estándar de librería GUI (interfaz de usuario gráfica) para Java SE pero previsiblemente ambos serán incluidos como tal en un futuro.

4.2.1.1 Disponibilidad

Las APIs de JavaFX están disponibles como una característica totalmente integrada de Java SE Runtime Environment (JRE) y de Java Development Kit (JDK). Ya que el kit de desarrollo en Java (JDK) está disponible para la mayoría de plataformas (Windows, Mac OS X y Linux), las aplicaciones de JavaFX compiladas para JDK 7 y versiones posteriores pueden ejecutarse en la mayoría de plataformas de escritorio. El apoyo para las plataformas

ARM ha sido posible con JavaFX 8. JDK para ARM incluye la base, gráficos y controles de componentes de JavaFX.

4.2.1.2 ¿Qué puedo desarrollar con JavaFX?

Con JavaFX se puede desarrollar muchos tipos de aplicaciones. Normalmente, dichas aplicaciones suelen trabajar en red y estar desplegadas a lo largo de múltiples plataformas mostrando información en una moderna interfaz de usuario de alto rendimiento que soportan audio, vídeo, gráficos y animaciones.

4.2.2 JavaFX Scene Builder 2.0

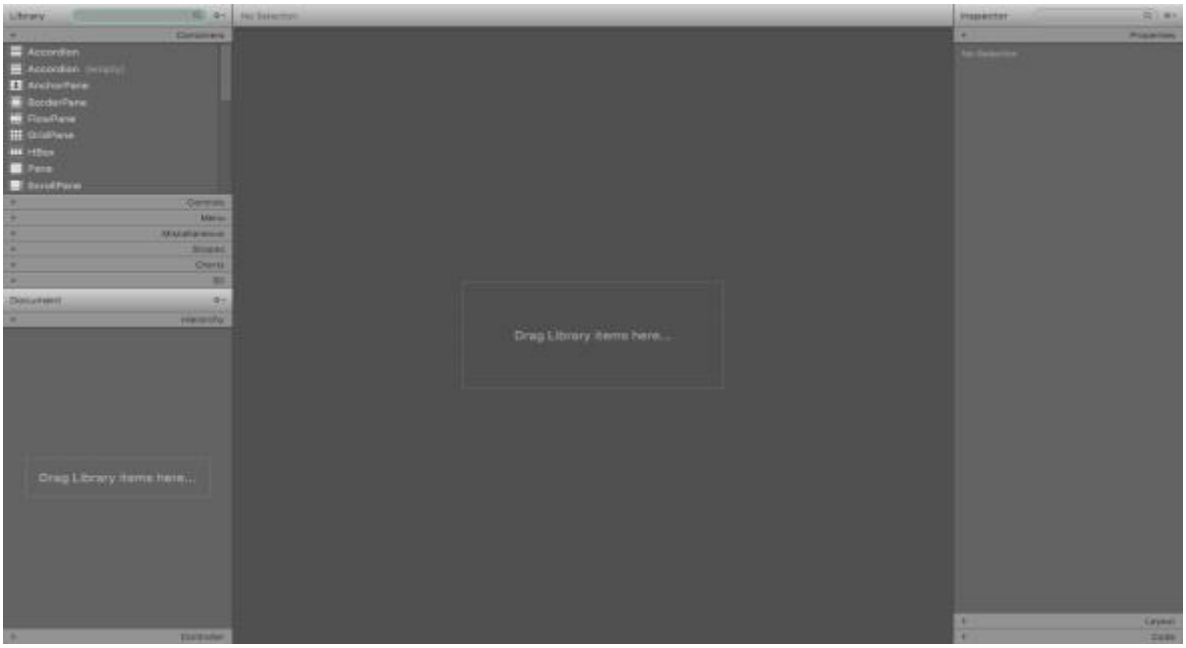


Imagen 13 - Scene Builder

Es una herramienta visual para aplicaciones JavaFX que permite desarrollar interfaces de usuario de aplicaciones JavaFX sin código. Los usuarios pueden hacer click y arrastrar los componentes hacia el área de trabajo, modificar sus propiedades, aplicar hojas de estilo y el código del archivo FXML que estamos usando se actualizará automáticamente en segundo

plano. El resultado es dicho archivo FXML que puede ser combinado con un proyecto Java mediante la unión de la interfaz de usuario con la lógica de la aplicación.

4.2.3 JDK 8

Java Development Kit (JDK) es una de las implementaciones de Java EE, Java SE o Java ME publicadas por Oracle Corporation en forma de software binario cuyo objetivo son los desarrolladores de Java en Solaris, Linux, Mac OS X o Windows. El JDK incluye una JVM (Java Virtual Machine) y otros recursos para finalizar el desarrollo de una aplicación Java.

Desde los inicios de la plataforma Java, esto ha sido, con mucha diferencia, el software de desarrollo (SDK) más utilizado. El 17 de noviembre Sun anunció que lo iban a publicar bajo una licencia general pública GNU (GPL), convirtiéndolo, por tanto, en software libre.

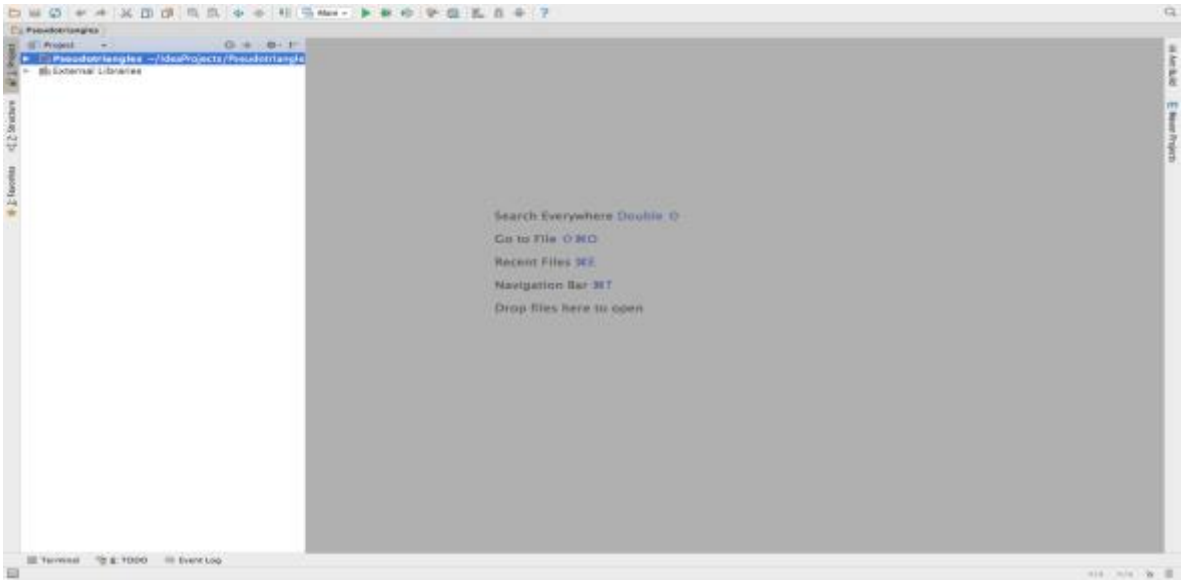


Imagen 14 - IntelliJ IDEA

Esto sucedió mayormente durante el 8 de mayo de 2007, cuando Sun aportó el código fuente a OpenJDK.

Para el desarrollo del proyecto, se ha usado específicamente la última versión de Java SE ya que incluye un JRE completo además de herramientas para el desarrollo, depurar y para monitorizar aplicaciones java.

4.2.4 IntelliJ IDEA

Es un entorno de desarrollo integrado (IDE) para el desarrollo de programas informáticos. Cabe destacar que fue desarrollado por JetBrains, antes conocido como IntelliJ, y que no está basado en Eclipse.

La primera versión fue publicada en enero del 2001 y fue una de los primeros entornos de desarrollo integrado para Java con una navegación de código avanzado y con capacidades de refactorización de código. En 2010 recibió de Inforworld la nota más alta y lo reconocía como la mejor herramienta de programación superior de Java por encima de otro entornos como Eclipse, NetBeans y Oracle JDeveloper. Por último, en 2014 Google anunció la versión 2.1 de Android Studio basado en el código abierto en la edición comunitaria de IntelliJ IDEA.

4.3 Desarrollo de la aplicación

La aplicación ha sido desarrollada en Java utilizando las tecnologías descritas en la sección previa. Gracias a JavaFX, podemos descomponer el proyecto en 3 partes.

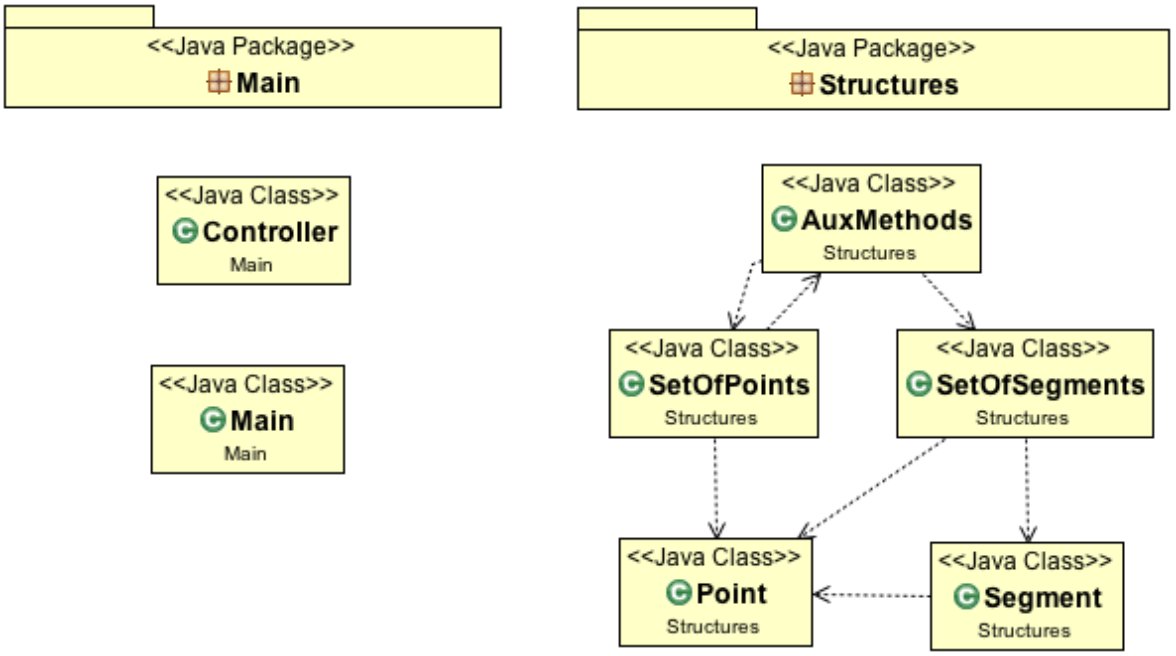
Por un lado tenemos la clase .fxml dónde se estará el código de la interfaz de usuario. Esta clase la explicaremos con detalle más adelante.

Por otro, tenemos las clases java Main y Controller las cuales hacen de intermediarios entre la implementación del programa y su interfaz o, dicho de otra forma, entre backend y frontend.

Finalmente, se ha creado un paquete llamado Structures dónde se van a localizar todas las clases pertenecientes a la lógica de la aplicación.

4.3.1 Diseño de bajo de nivel

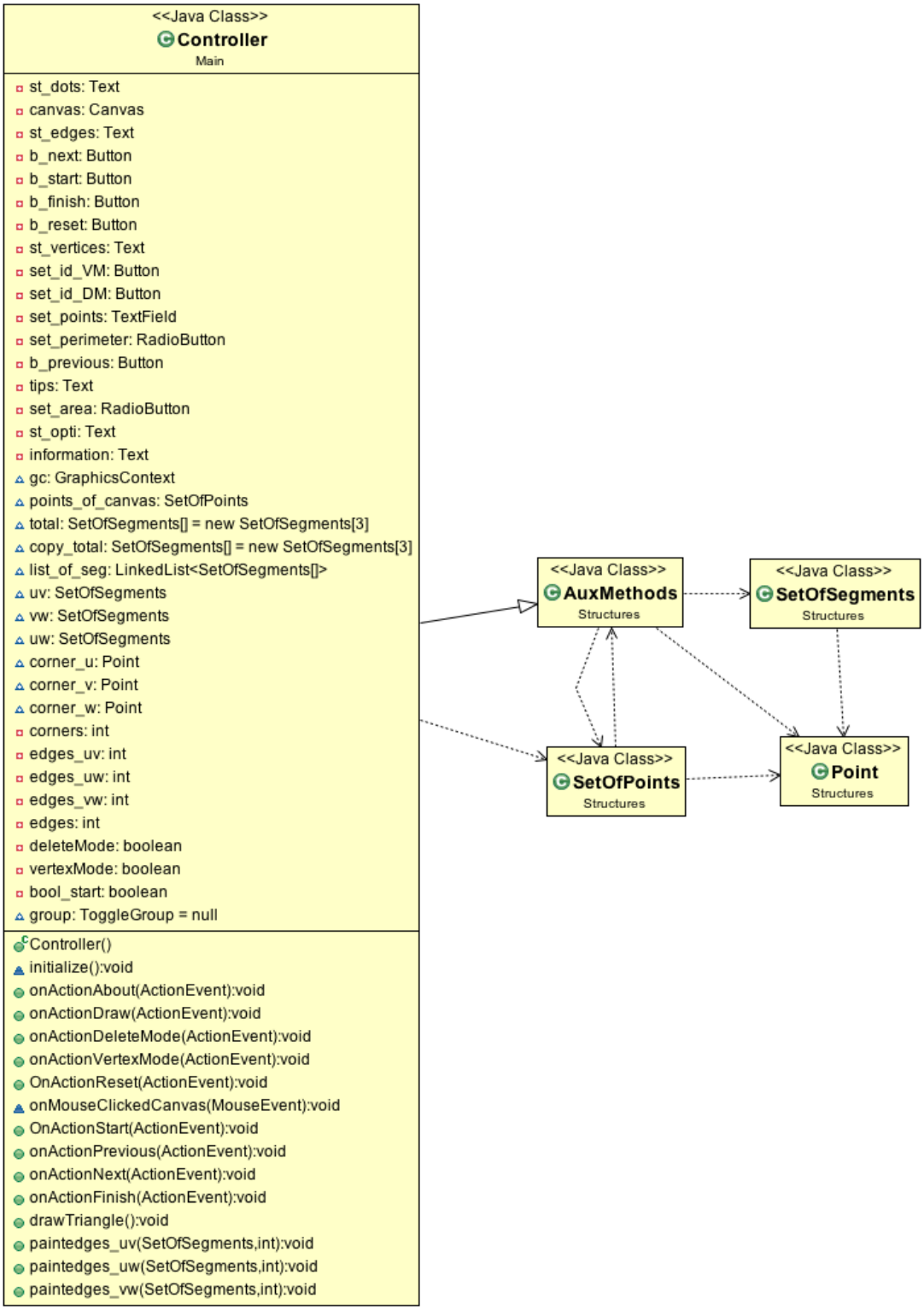
La estructura general del proyecto es la siguiente:



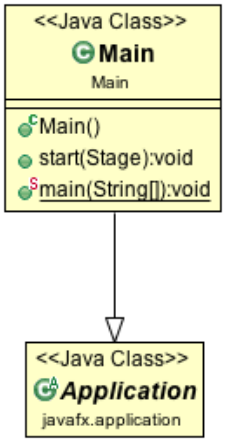
En el paquete Main tenemos las clases Controller.java, Main.java y simple.fxml (no mostrada en el diagrama ya que no es java, es decir, no es relevante para esta sección). La primera interactúa con la lógica del programa y la segunda ejecuta y muestra la interfaz de usuario.

En el paquete Structures tenemos varios métodos. En primer lugar tenemos la clase AuxMethods.java dónde se alojarán todos los algoritmos importantes. Después, tenemos las estructuras de datos utilizadas. La clase Point.java es la raíz de todas las clases y consiste en un punto de coordenadas x e y. Un segmento de la clase Segment.java va a consistir en dos puntos: origen y fin. SetOfPoints es una clase que contiene un HashMap cuya clave es la posición del punto en el HashMap y el valor el punto en sí. SetOfSegments es lo mismo que SetOfPoints pero con segmentos. Todas estas clases han sido creadas para facilitar ciertas operaciones necesarias como mantener un conjunto de puntos contenidos en un triángulo (en SetOfPoints) o pintar segmentos en el lienzo de la aplicación (organización facilitada por SetOfSegments).

4.3.1.1 Controller.java

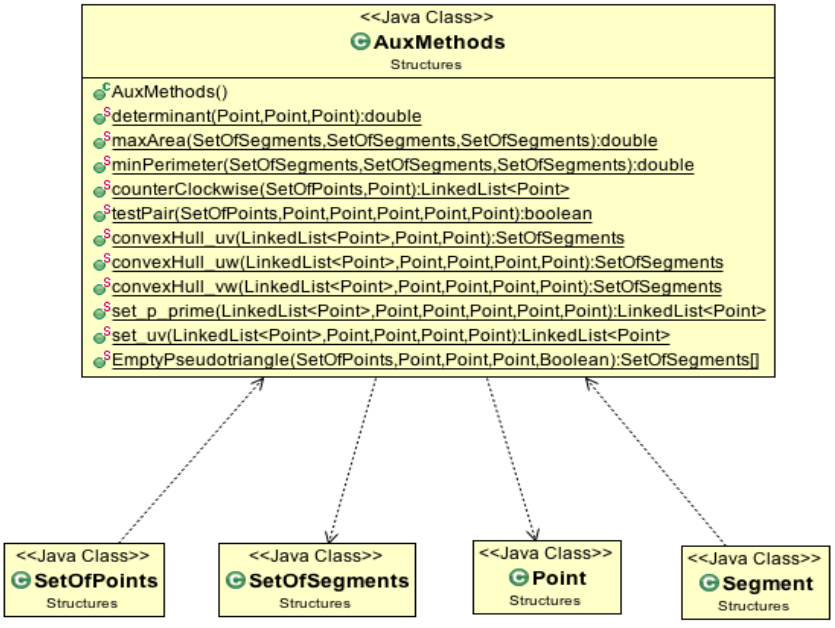


4.3.1.2 *Main.java*



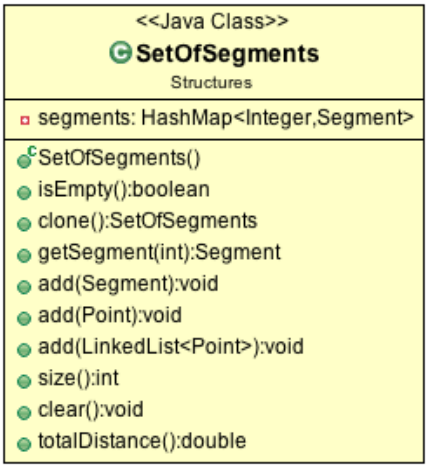
Como podemos observar, esta clase sólo interactúa con el frontend. Arranca la interfaz de usuario.

4.3.1.3 *AuxMethods.java*

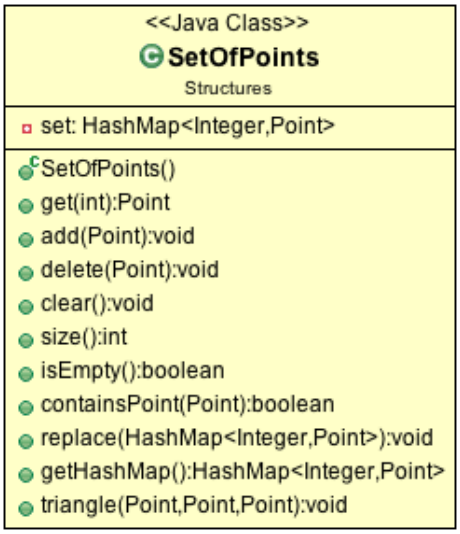


Aquí podemos observar cómo esta clase implementa todos los algoritmos y hace uso de las estructuras de datos creadas.

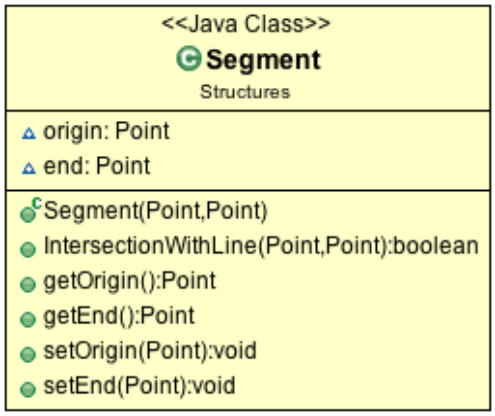
4.3.1.4 SetOfSegments.java



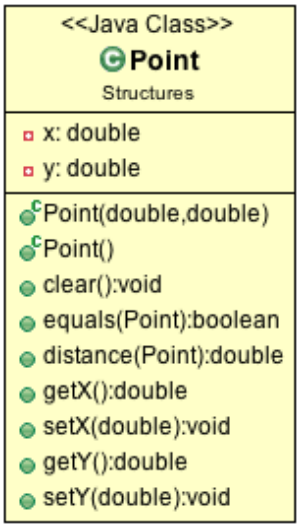
4.3.1.5 SetOfPoints.java



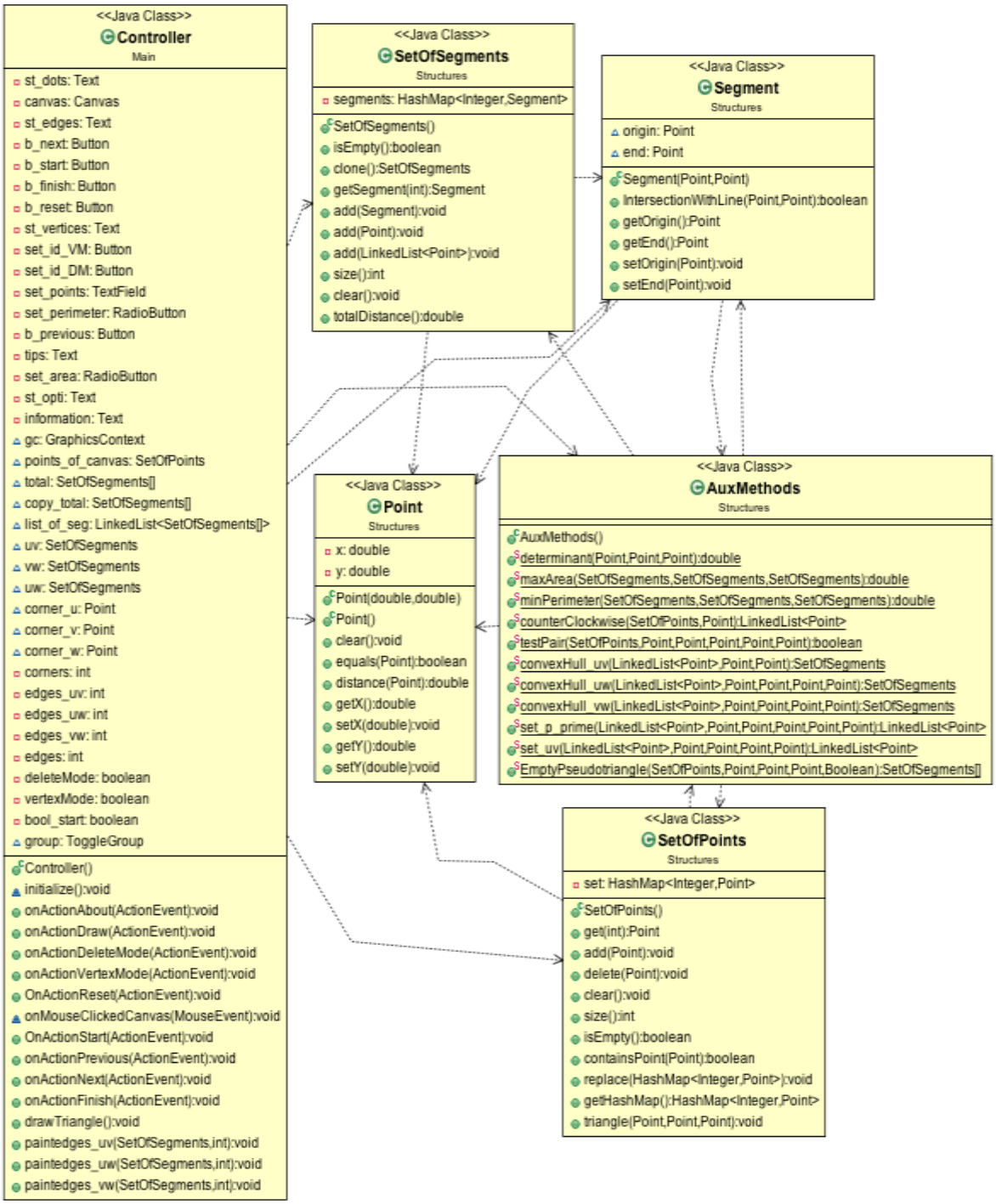
4.3.1.6 *Segment.java*



4.3.1.7 *Point.java*



4.3.1.8 *Vista general*



4.3.2 Implementación de la aplicación

Cabe destacar que se ha usado la clase Point.java personalizada en lugar de la existente por defecto por el hecho que Point.java está modificada. Se ha añadido un método que llamado distance() dónde devolverá la distancia Euclídea entre dos puntos.

```
/**
 * This method will tell the distance between two points.
 * @param //p1
 * @return
 */
public double distance(Point p1){
   return Math.sqrt(Math.pow((this.x  - p1.getX()),2) + Math.pow((this.y - p1.getY()),2));
}
```

Por otro lado, en la clase Segment.java está implementado un método llamado IntersetionWithLine() el cual devuelve true si la línea formada por los dos puntos pasados como parámetro intersectan con el segmento desde el que se llama. False si no.

```
/**
 * True if the line that contains ab will cross the segment (this).
 * Point w is the third corner.
 *
 * @param a
 * @param b
 * @return
 */
public boolean IntersectionWithLine(Point a, Point b){

   // Segment PjV
   if (AuxMethods.determinant(a, this.origin, b) > 0)
      if (AuxMethods.determinant(a, this.end, b) < 0)
         if ((AuxMethods.determinant(this.origin, this.end, b) > 0) || (AuxMethods.determinant(this.origin,
this.end, b) < 0)) // double check
            return true;

   return false;
}
```

SetOfPoints contiene un método especial el cual es triangle() dónde eliminará todos los puntos que estén fuera del triángulo que forma el conjunto de puntos según los tres vértices convexos que se pasen como parámetro.

```
/**
 * Only points inside of the triangle consisted on u, v and w will remain.
 * Also no three points will be lineal neither with the corners nor with others.
 * @param u
 * @param v
 * @param w
 */
```

```
public void triangle(Point u, Point v, Point w){

    HashMap<Integer, Point> newHM = new HashMap<>();

    /**
     * First we make sure there is not lineal dependency among them
     */
    HashMap<Integer, Point> lineal = new HashMap<>(); // we will store here the points that are lineal with
others
    int s = this.set.size();
    int counter=0;

    // Find points that are lineal
    for (int i = 0; i<s-2; i++){
        for (int j = i+1; j<s-1; j++){
            for (int k = i+2; k<s; k++){
                if (AuxMethods.determinant(this.set.get(i), this.set.get(j), this.set.get(k)) == 0){
                    if (lineal.containsValue(this.set.get(k))){
                        lineal.put(counter, this.set.get(k));
                        counter++;
                    }
                }
            }
        }
    }

    // Copy in newHM with no lineal points
    counter=0;
    for (int i=0; i<this.set.size() && !lineal.containsValue(this.set.get(i)); i++){
        newHM.put(counter, this.set.get(i));
        counter++;
    }

    // Update set
    this.set.clear();
    this.set = (HashMap) newHM.clone();
    newHM.clear();

    //

    /**
     * We make sure they are not lineal with the corners and build the triangle.
     */
    s = this.set.size(); // size > 2

    counter = 0;
    for (int i=0; i<s; i++){
        // If w lies above the segment uv
        if (AuxMethods.determinant(u, v, w) > 0){
            if (AuxMethods.determinant(u, v, this.set.get(i)) > 0){ // Oriented positively with respect to uv segment
                if (AuxMethods.determinant(v,w, this.set.get(i)) > 0){ // Oriented positively with respect to vw
segment
                    if(AuxMethods.determinant(w,u, this.set.get(i)) > 0){ // Oriented positively with respect to wu
segment
                        newHM.put(counter, this.set.get(i));
                        counter=counter+1;
                    }
```

```
            }
        }
    }

    // If w is located below the segment uv
    else {
        if (AuxMethods.determinant(u,v, this.set.get(i)) < 0){ // Oriented negatively with respect to uv segment
            if (AuxMethods.determinant(v,w, this.set.get(i)) < 0){ // Oriented negatively with respect to vw
segment
                if(AuxMethods.determinant(w,u, this.set.get(i)) < 0){ // Oriented negatively with respect to wu
segment
                    newHM.put(counter, this.set.get(i));
                    counter =counter+1;
                }
            }
        }
    }
  }

  this.set.clear();
  this.set = (HashMap) newHM;

}
```

Este método primero comprueba que no existen 3 puntos linealmente independientes y luego forma el triángulo con los vértices pasados como parámetro.

4.3.3 Interfaz de usuario

JavaFX ha sido muy útil en el desarrollo de la interfaz de la aplicación ya que permite diseñarla y desarrollarla mediante *drag-and-drop*. Dicha interfaz se compone de varias secciones. Empezando por la izquierda, tenemos una caja llamada Status donde se mostrará el número de puntos en el lienzo, el número de vértices que conformarán el pseudotriángulo, el número de aristas y su optimización, es decir, el tamaño en centímetros cuando hablamos del perímetro y en centímetros cuadrados cuando hablamos del área.

Avanzando hacia abajo tenemos la sección de configuración. Aquí tenemos una caja dónde podemos insertar un número no superior a 300 para poder generarlos y dibujarlos. La creación de dichos puntos sigue una distribución normal.

A continuación tenemos dos botones donde podremos activar el modo de borrar puntos o activar el modo de dibujar los vértices convexos. Dichos botones son exclusivos, no se pueden activar a la vez por razones obvias.

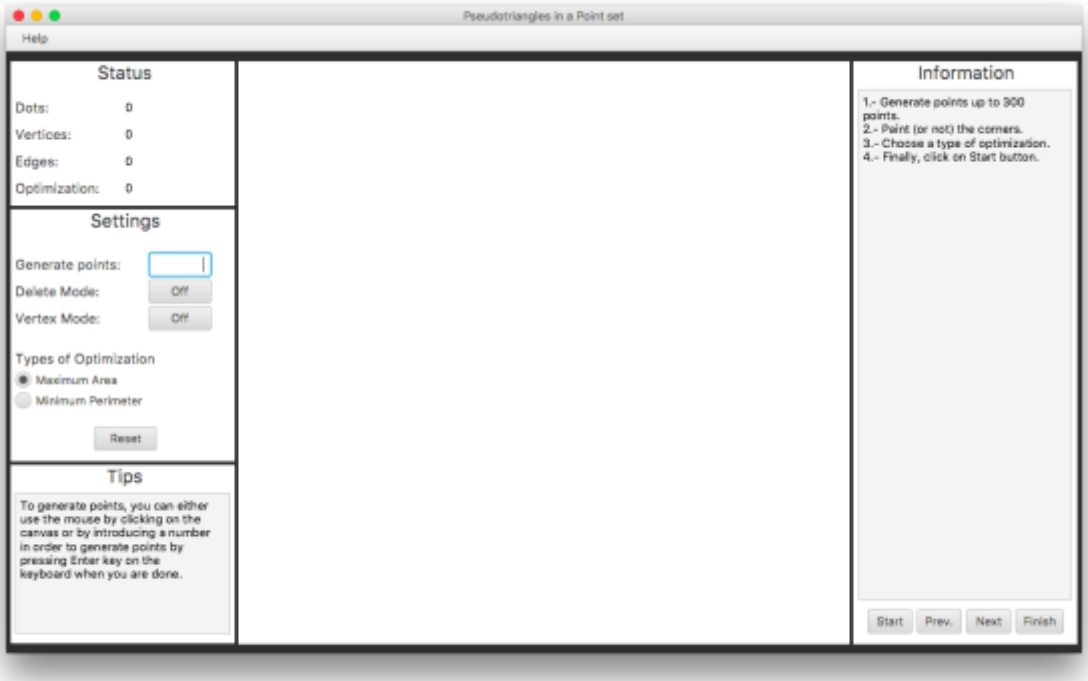


Imagen 15 - Aplicación Desarrollada

Ahora tenemos que elegir entre el tipo de optimización que queremos ya sea según su perímetro o según su área. Para terminar este apartado tenemos el botón Reset donde al hacer click en él borraremos todo lo que tenemos y podremos empezar de nuevo.

Finalmente en este lado tenemos la sección Consejos dónde en todo momento no irá dando pistas de qué podemos o tenemos que hacer. Es importante no sentirse perdido en ningún momento de la navegación en la aplicación.

En medio de la aplicación tenemos el lienzo donde tendrá lugar todo lo relativo a los dibujos.

Para terminar, en la parte derecha tenemos la sección de información dónde se proporcionarán datos didácticos para saber qué se está haciendo mientras se ejecuta el algoritmo. En su parte inferior, tenemos 4 botones dónde tenemos uno para ejecutar el algoritmo y el resto para la navegación durante el proceso.

4.3.4 Implementación de algoritmos

Este apartado hace referencia a la clase AuxMethods.java dónde se ha implementado la lógica importante del programa.

Empezamos con el método determinant() el cual nos dirá la orientación del tercer punto pasado como parámetro según los dos primeros.

```
/***
 * If > 0 -> positively oriented
 * If == 0 -> lineal
 * If < 0 -> negatively oriented
 */
public static double determinant(Point a, Point b, Point c){
    double ans = ((b.getX()*c.getY() + a.getX()*b.getY() + a.getY()*c.getX()) -
            (a.getY()*b.getX() + a.getX()*c.getY() + b.getY()*c.getX()));
    return ans;
}
```

A continuación tenemos el método maxArea() donde tendremos como resultado el área del polígono pasado como parámetro. Dicho polígono es un pseudotriángulo con 3 cadenas de segmentos. Para el cálculo del área se ha usado el método del determinante de Gauss.

```
/**
 * Gauss Area Formula (or shoelace formula / shoelace algorithm)
 * @param uv
 * @param uw
 * @param vw
 * @return
 */
public static double maxArea(SetOfSegments uv, SetOfSegments uw, SetOfSegments vw){
    double res = 0.0;
    LinkedList<Point> unifying = new LinkedList<>();

    //System.out.println("Tamaño inicial de uv: " + uv.size() + " uw: " + uw.size() + " vw: " + vw.size());

    //Unifying all the points in counterclockwise
    for (int i=0; i<uv.size(); i++){
        unifying.add(uv.getSegment(i).getOrigin());
    }

    for (int i=0; i<vw.size(); i++){ // last point of uv is first point of vw
        unifying.add(vw.getSegment(i).getOrigin());
    }

    //backwards
    for (int i=uw.size()-1; i>=0; i--){// last point of vw is "last" point of uw
        unifying.add(uw.getSegment(i).getEnd());
    }

    //unifying.add(uw.getSegment(0).getOrigin()); // first point of the polygon
    int j;
```

```
    for (int i = 0; i < unifying.size(); i++) {
        j = (i + 1) % unifying.size();
        //area += points[i].x * points[j].y - points[j].x * points[i].y;
        res = res + unifying.get(i).getX()*unifying.get(j).getY()  - unifying.get(j).getX()*unifying.get(i).getY();
    }

    //System.out.println("Tamaño final de unifying:" + unifying.size());

    if (res<0)
        res=res*(-1.0);// absolute value

    return res / 2.0;
}
```

Para calcular el perímetro, se ha llamado recursivamente al método implementado en Point.java que se encarga de medir la distancia Euclídea entre dos puntos.

```
/**
 * Euclidean Distance
 * @param uv
 * @param uw
 * @param vw
 * @return
 */
public static double minPerimeter(SetOfSegments uv, SetOfSegments uw, SetOfSegments vw){

    double res = uv.totalDistance() + uw.totalDistance() + vw.totalDistance();

    return res;
}
```

Ahora tenemos el método que se encarga de ordenar los puntos contenidos dentro de un pseudotriángulo en sentido contrario a la agujas del reloj según w.

```
/**
 * Points will be sorted counter-clockwise around w
 * @param w
 * @return
 */
public static LinkedList<Point> counterClockwise(SetOfPoints set, Point w){
    LinkedList<Point> list = new LinkedList<Point>();
    int size = set.size();

    // Add a point to start with
    if (list.isEmpty()) // if beginning
        list.add(set.get(0));

    for(int i=1; i<size; i++) {
        boolean inserted = false;
        ListIterator iterator = list.listIterator();
        while(iterator.hasNext() && !inserted){
            // if positive, it goes after the current point
```

```
            Point pe = (Point) iterator.next();
            if (determinant(w, pe, set.get(i)) < 0) {
                if ((i-1) == 0){
                    list.addFirst(set.get(i));
                    inserted = true;
                }
                else{
                    list.add(i-1, set.get(i)); // if negative, it goes before
                    inserted = true;
                }
            }else{
                if (!iterator.hasNext()){ // If end of list. Check this AGAIN
                    list.add(set.get(i));
                    inserted = true;
                }else{
                    continue;
                    //iterator.next();
                }
            }
        }
    }
    return list;
}
```

A continuación tenemos uno de los métodos clave de este proyecto: TestPair. Dicho algoritmo se encarga de conseguir los pi y pj siguiendo una serie de restricciones.

```
/**
 * TestPair Algorithm
 * @param set
 * @param u
 * @param v
 * @param w
 * @param pi
 * @param pj
 * @return
 */
public static boolean testPair(SetOfPoints set, Point u, Point v, Point w, Point pi, Point pj){

    //if l(w,pi) intersection vpi != 0 OR l(w,wpj) intersection upi != 0
    Segment upi = new Segment(u, pi);
    Segment pjv = new Segment(pj, v);

    if (pjv.IntersectionWithLine(w,pi) || upi.IntersectionWithLine(w,pj))
        return false;

    for (int i=0; i<set.size(); i++){
        // if l(u,pi)- intersection l(v,pj)+ intersection l(w,pi)- intersection set
        if (determinant(u, pi, set.get(i)) < 0) // below upi
            if (determinant(pj, v, set.get(i)) > 0) // above pjv
                if (determinant(w, pi, set.get(i)) < 0) // to the left of wpi
                    return false;

        // if l(u,pi)+ intersection l(v,pj)- intersection l(w,pj)+ intersection set
```

```
        if (determinant(u, pi, set.get(i)) > 0) // above upi
            if (determinant(pj, v, set.get(i)) < 0) // below pjv
                if (determinant(w, pj, set.get(i)) > 0)
                    return false;
    }

    return true;
}
```

Los métodos que se encargan de generar el cierre convexo son convexHull_xx(). Son tres métodos que construirán en cada momento la envolvente conexa cuando sean llamados por el método EmptyPseudotriangle que implementa el algoritmo ShortestPseudotriangle.

Después, tenemos los métodos set_uv() y set_p_prime(). En el primero separamos del conjunto de puntos P todos los puntos que se encuentren por debajo de las líneas que forman los segmentos $\overline{up_i}$ y $\overline{vp_j}$. En contraposición, el segundo método se encargará de coger el conjunto de puntos restante, es decir, los puntos que esté por encima del primer o del segundo segmento.

Finalmente tenemos el algoritmo principal implementado en el método EmptyPseudotriangle que está a continuación. Debido a su tamaño, sólo se mostrará parte de él.

```
// Sort P counterclockwise in angular order around w
LinkedList<Point> angular_order = new LinkedList<>(counterClockwise(set, w));
//System.out.println("Empty Pseudotriangles - El número de puntos ordenados en sentido contrario a las agujas del reloj es: " + angular_order.size());

// for all choices of pi and pj for which Test-Pair returns TRUE
for (int i=0; i<set.size(); i++){
    for (int j=0; j<set.size(); j++){
        if (j==i) continue;
        if (testPair(set, u, v, w, set.get(i), set.get(j))){
            // Select the subset of points belowl(u,pi) and belowl(v,pj) and compute Cuv
            LinkedList<Point> set_of_uv= new LinkedList<>(set_uv(angular_order, u, v, set.get(i), set.get(j)));
            SetOfSegments aux_UV4 = convexHull_uv(set_of_uv, u, v);

            // Determine P'
            LinkedList<Point> p_prime = new LinkedList<>(set_p_prime(angular_order, u, v, w, set.get(i), set.get(j)));

            //if (!p_prime.isEmpty()){
            //System.out.println("Empty Pseudotriangles - Pi y Pj están contenidos dentro de P'.");
            //System.out.println("Tamaño de p_prime: " + String.valueOf(p_prime.size()));
            //System.out.println("");

            for (int y=0; y<p_prime.size()-1; y++){
                if (p_prime.get(y).getX() < p_prime.get(y+1).getX()){
                    SetOfSegments aux_UW4 = convexHull_uw(p_prime, u, w, set.get(i), p_prime.get(y));
```

```
                SetOfSegments aux_VW4 = convexHull_vw(p_prime, v, w, set.get(j), p_prime.get(y+1));
                //System.out.println("El valor x del punto y es: " + String.valueOf(p_prime.get(y).getX()));
                //System.out.println("El valor x del punto y+1 es: " + String.valueOf(p_prime.get(y+1).getX()));
                //System.out.println("");

                if (optimization){ // max area
                    d = maxArea(aux_UV4, aux_UW4, aux_VW4);
                    if (d < total_size){
                        total[0] = aux_UV4; total[1] = aux_UW4; total[2] = aux_VW4;
                        total_size = d;
                    }
                }else { // min perimeter
                    d = minPerimeter(aux_UV4, aux_UW4, aux_VW4);
                    //System.out.println("El valor parcial de d es: " + String.valueOf(d));

                    if (d > total_size){
                        total[0] = aux_UV4; total[1] = aux_UW4; total[2] = aux_VW4;
                        total_size = d;
                        //System.out.println("El valor nuevo de total_size es: " + String.valueOf(total_size));
                    }
                }
            }
        }
    }// testPair
  }
}// for i
```

Como se puede observar, primero se ordenan los puntos en sentido contrario a las agujas del reloj y se van consultando todos los pseudotriángulos para los cuales TestPair devuelve true.

4.4 Manual de usuario

Debido a que el programa ya implementa una amplia serie de instrucciones, en este apartado no se explicará con mucho detalle el funcionamiento del mismo. Sólo las nociones básicas e importantes que son necesarias para poder afrontar su procesamiento.

En primer lugar debemos generar un conjunto de puntos bien con el ratón o bien con la opción que se ha proporcionado de crear un conjunto de puntos siguiendo una distribución normal. A partir de aquí podemos borrar puntos o generar los tres vértices convexos que nos servirán para la triangulación. Como sabemos, el dibujar dichos vértices es opcional.

En cualquier momento podremos presionar el botón reset para empezar de nuevo pero antes de este botón tenemos las dos opciones a elegir: el tipo de optimización que se quiere realizar.

A partir de aquí, presionamos el botón Start para procesar el algoritmo y después los botones de navegación Prev., Next o Finish para mostrar los diferentes pasos por los que atraviesa el algoritmo.

En todo momento tendremos información proporcionada por las secciones Tips o Information.

4.5 Problemas encontrados

El único problema encontrado es que se encontró un fallo en la descripción del algoritmo Rooted-Tree así que se ha optado por una construcción alternativa, la cual no hizo precisamente reducir el tiempo de ejecución quedándose finalmente en $O(n^4)$. Esto se debe a que el árbol que se usa en el algoritmo original almacena el tamaño de su arista en cada nodo por lo que al recorrer dichos puntos, dicha consulta no depende de la ejecución en sí del árbol. Sólo depende de la consulta en los determinados puntos mientras que mi solución propuesta sí que recorre todos los puntos y genera una cadena según el par de puntos pi y pj encontrados.

5 PRUEBAS

5.1 Generación de un pseudotriángulo optimizando su perímetro

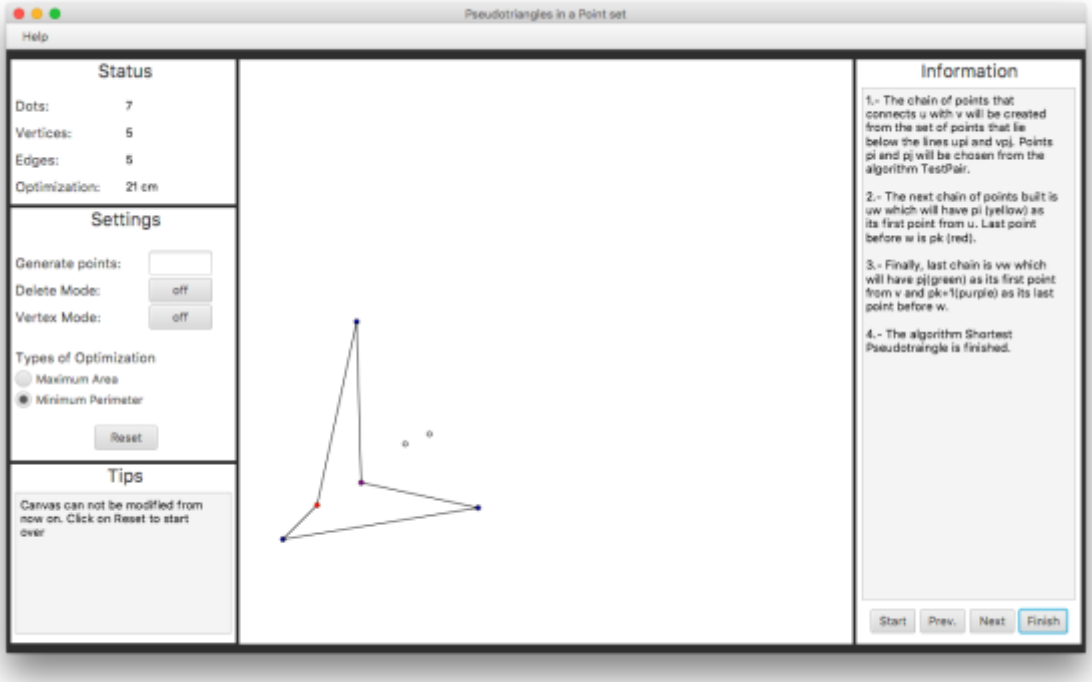


Imagen 16 - Prueba 1

El caso de prueba ha sido generado con éxito.

5.2 Generación de un pseudotriángulo optimizando su área

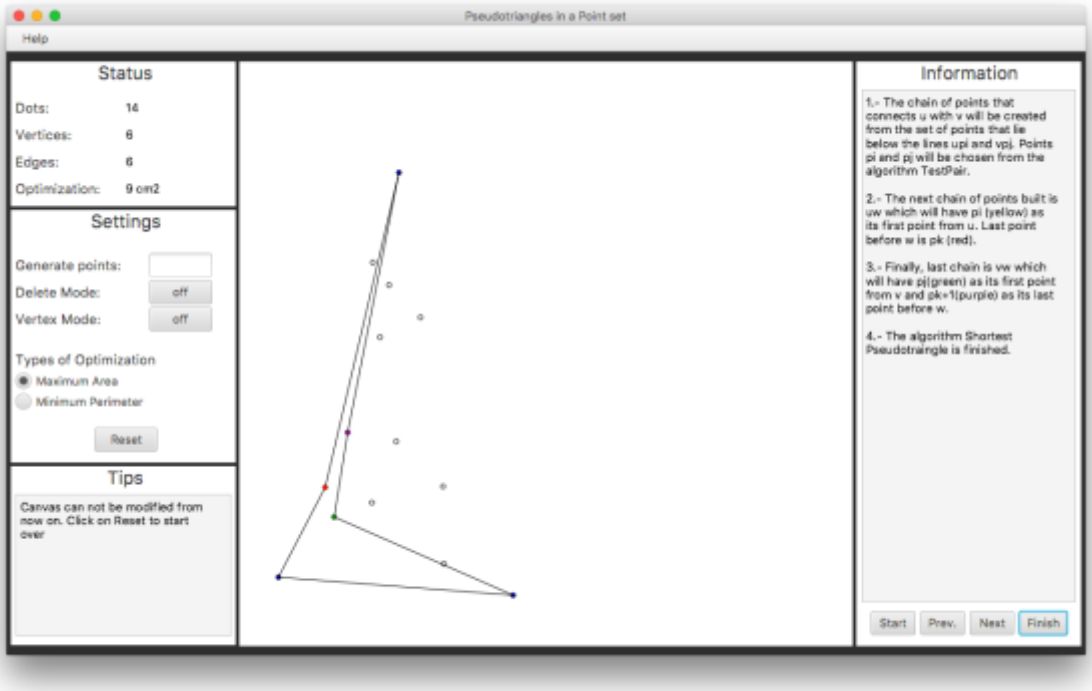


Imagen 17 - Prueba 2

El caso de prueba ha sido ejecutado con éxito.

6 CONCLUSIONES

Este trabajo me ha resultado muy interesante ya que las matemáticas son un tema que me interesa a nivel personal. Por otra parte, también la informática así que me ha parecido una mezcla perfecta que me ha permitido desarrollar mis aptitudes analíticas ya que dicho trabajo fue extraído de un paper que se dio a conocer en una conferencia de geometría computacional en Canadá.

7 REFERENCIAS

- Empty pseudo-triangles in point sets – Hee-Kap Ahn, Sang Won Bae, Marc van Kreveld, Iris Reinbacher, Bettina Speckman.
- Optimal Empty Pseudo-Triangles in a Point Set – Hee-Kap Ahn, Sang Won Bae, Iris Reinbacher.

8 BIBLIOGRAFÍA

- Determinante de Gauss - https://www.wikiwand.com/en/Shoelace_formula
- Distancia Euclídea - https://www.wikiwand.com/en/Euclidean_distance
- Tutorial de Scene Builder - https://blogs.oracle.com/jmxetc/entry/connecting_scenebuilder_edited_fxml_to
- Tutorial de JavaFX - http://docs.oracle.com/javase/8/javase-clienttechnologies.htm
- JavaFX - https://en.wikipedia.org/wiki/JavaFX
- IntelliJ IDEA - https://es.wikipedia.org/wiki/IntelliJ_IDEA
- Scene Builder - https://docs.oracle.com/javase/8/scene-builder-2/get-started-tutorial/index.html
- Generació de números aleatorios en Java - http://puntocomnoesunlenguaje.blogspot.com.es/2012/07/generacion-de-numeros-aleatorios-en-java.html
- Geometría de los determinantes - http://recursostic.educacion.es/descartes/web/materiales_didacticos/determinantes/geo_det.pdf

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Jul 08 13:40:52 CEST 2016
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)